Electronic Notes in Theoretical Computer Science 89 No. 3 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume89.html 19 pages

Space-Reduction Strategies for Model Checking Dynamic Software

Robby [a], Matthew B. Dwyer [a], John Hatcliff [a] and Radu Iosif [b]

[a] *Department of Computing and Information Sciences, Kansas State University*
234 Nichols Hall, Manhattan KS, 66506, USA.
{robby,dwyer,hatcliff}@cis.ksu.edu

[b] *Verimag/CNRS*
2 Avenue de Vignate, 38610 Gieres, France.
Radu.Iosif@imag.fr

Abstract

Effective model-checking of modern object-oriented software systems requires providing support for program features such as dynamically created threads, heap-allocated objects and garbage collection. These features have often proven problematic to treat using many previous model-checking frameworks that do not provide sophisticated heap representations and optimizations.

In this paper, we define a flexible framework for combined heap and thread symmetry reductions in explicit-state model checking that can be tuned to trade run-time overhead for precision. In addition, we describe various strategies for duplication-reducing state-space encodings for object-oriented heap structures. We have implemented these techniques in Bogor (our extensible software model-checking framework), and we present empirical data to support the effectiveness of these memory reductions on a collection of realistic examples and to demonstrate that they improve upon previous approaches. These techniques, formalized in a group theoretic framework, can be applied to any non-deterministic heap object diagram.

1 Introduction

Despite its significant complexity, model checking has proven to be an effective technique for uncovering subtle errors in the implementation of concurrent programs [2]. Researchers are working on a variety of techniques to allow model checking to scale to be an effective analysis framework for large, complex concurrent programs. Two directions being pursued are automated data abstraction (e.g., [15]) and thread modular approaches (e.g., [11,14]). While these offer the potential for scalability, the use of such techniques is currently

limited to reasoning about properties of scalar data that are local to threads. They are not, for example, applicable to reasoning about properties that span multiple program threads and that relate to complex heap structures. For those kinds of properties, which arise in specifications of object-based concurrent systems, general explicit-state model checking approaches for dynamic software [2,6,7] must be used.

In this paper, we present methods that enable memory reductions for explicit-state model checking of dynamic software. Reduction of both the number of program states that are stored and of the size of those states are presented. Our work extends existing approaches for exploiting symmetries in the structure of a program's dynamic data (i.e., heap) [17]. Specifically, we develop a flexible framework for identifying symmetric heap states that is tunable to trade run-time overhead for precision. In addition, we improve the thread (or process) symmetry [1] in the calculation of heap symmetry information to improve on the results of [17] by providing better heuristics for thread ordering. In general, our approach can be applied to any non-deterministic heap object diagram. The framework is presented using group-theoretic notions that greatly simplify its formalization. We also extend existing approaches to compressing the representation of individual program states (e.g., [16]) by identifying information that is shared by multiple state components and eliminating that duplicate information from the state encoding.

The contributions of the paper include: (*i*) the definition of a framework and algorithm for combined heap and thread symmetry reduction in explicit-state model checking; (*ii*) the definition of a framework and algorithm for duplication-reducing state-space encodings; and (*iii*) empirical data to support the effectiveness of these memory reductions on a collection of realistic examples.

These reductions are implemented in a new model checking framework called *Bogor*. Bogor [21] is an extensible and highly modular explicit-state model checking framework. It provides a rich modeling language including features that allow for dynamic creation of objects and threads, garbage collection, virtual method calls and exception handling. It also provides extension mechanisms to ease the task of customizing the model checker with domain-specific abstraction layers and to accommodate, for example, variations in scheduling policies, search modes for state exploration, state encodings, and checkers for specification languages. Bogor has been customized to support different kinds of software artifacts including: Java source code [9] and event-driven component-based designs [8,12]. The techniques described in this paper are broadly applicable and help to make Bogor an efficient core on which to build domain-specific model checking-based analyses.

The next section presents an example used to introduce the reductions presented in the paper. Section 3 presents our general symmetry reduction framework and describes how heap and thread symmetries are combined. Section 4 describes how information from symmetry reductions can drive the

```
system nDiningPhilosophers {
  top record Object {}
  record Fork extends Object { boolean isHeld; }
  thread Philosopher(Fork left, Fork right) {
    loc loc0: // get left
      when !left.isHeld do {
        left.isHeld := true;
      } goto loc1;
    loc loc1:  // get right
      when !right.isHeld do {
        right.isHeld := true;
      } goto loc2;
    loc loc2: // drop right
      when true do {
        right.isHeld := false;
      } goto loc3;
    loc loc3: // drop left
      when true do {
        left.isHeld := false;
      } goto loc0;
  }
  main thread MAIN() { ... }
}
```

Fig. 1. N-Dining Philosophers Example (excerpts)

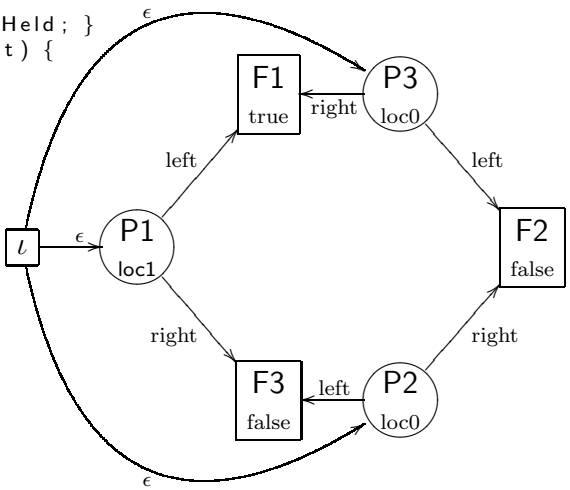

Fig. 2. 3-Dining Philosophers State Example

compression of heap data encodings. Section 5 presents preliminary empirical results that suggest the benefit of using our memory reductions. Related work is discussed in Section 6 and we conclude with a discussion of future work in Section 7.

2 An Example

Figure 1 presents a fragment of the BIR (Bogor's input language) model for the N-dining philosophers problem – an example of resource deadlock in concurrent programs. This example is small, but interesting enough to illustrate many of our proposed reduction strategies.

A BIR system (in this case named `nDiningPhilosophers`) consists of declarations of records and threads. BIR records are used to model the state of Java objects, and a subtype relation is associated with them to model Java's class inheritance hierarchy. Accordingly, in our example, there is a unique *top* record named Object, and a Fork record that extends the Object record. The state of a fork is modeled by a boolean value: true if it is held and false otherwise. All non-top record types are required to extend some other record. When a record A extends another record B, then all of the fields of A are implicitly present in record B.

A thread declaration begins with the name of the thread followed by the declaration of any parameters. For example, the Philosopher thread takes two parameters of type Fork named left and right. The body of a thread consists of a sequence of *location* declarations. Thread execution begins at the first declared location. Each location construct consists of declarations

of *transformations* (transitions out of that location). A transformation is a guarded command and its execution is atomic. A *when* transformation consists of the definition of the guard and the command that may be executed when the guard holds. When several transformations are enabled at the same time, one is chosen for execution non-deterministically. A transformation declaration ends with a jump to a destination location or a *return* (which terminates the thread's execution). Due to space constraints, we elided the content of the main method which creates the initial state.

Figure 2 illustrates the state where there are 3 philosophers with 3 forks and after the first philosopher takes its left fork and the other two are ready to take their left forks. The location of each philosopher and the value of each fork's isHeld field are listed inside the circles and boxes, respectively.

3 Heap and Thread Symmetry Reductions

Symmetry reductions exploit the structure of states in order to identify equivalence classes. Only one state in each symmetry equivalence class needs to be visited, thus, it can significantly reduce the space and, because of that, the time required to verify a system. Assuming that states are ordered sets of components (i.e., objects, threads), the intuition behind these techniques is that the order in which these components are represented does not influence the future behavior of the system. Hence, identifying symmetries amounts to identifying bisimulations [13]. The larger the equivalences classes resulting from the symmetry, the more effective the reduction. Ideally, we would like to identify all possible symmetric states in a system on-the-fly (i.e., before generating the actual successors of the current state). The general problem of finding all symmetric states given a state, however, has been shown to be as hard as the *graph isomorphism problem* [3]. The latter problem is in NP in general and in P for directed deterministic graphs. A closely related problem, the *subgraph isomorphism problem* is known to be NP-complete.

The rest of this section is organized as follows: in Subsection 3.1 we introduce the formal concepts needed to reason about symmetry and in Subsection 3.2 we present a theoretical framework for developing symmetry reductions. Two instances of this framework, namely heap and thread symmetry are considered in Subsection 3.3. An algorithm implementing the latter is given in Subsection 3.4.

3.1 Preliminaries

Let us assume that the system we want to verify is given as a set of *states* denoted by $\mathcal{S}$. Each state $s \in \mathcal{S}$ is a finite set of *objects* (i.e. $s = \{o_i\}_{i \in I}$ indexed by the set I). For the moment we will ignore the connections between objects as well as other structural and typing information; this will be added later. Let G_I denote the set of all bijective functions (permutations) on I. It

can be shown that $(G_I, \circ, \mathcal{I})$ forms a group, with $\circ$ being function composition and $\mathcal{I}$ the identity permutation. For each permutation $\pi \in G_I$ we denote the *application* of the permutation to a state s by $\pi(s) = \{o_{\pi(i)}\}_{i \in I}$ (i.e., we change the index labeling on each object). Note that this operation induces a permutation group on states. We are now ready to formalize the general notion of *state symmetry*.

Definition 3.1 Given G a subgroup of G_I, we say that two states $s, t \in \mathcal{S}$ are G-symmetric ($s \approx_G t$) if and only if there exists a permutation $\pi \in G$ such that $\pi(s) = t$.

From basic group properties, we obtain that $\approx_G$ is an equivalence relation on $\mathcal{S}$, thus inducing a partition of $\mathcal{S}$ in equivalence classes. In order to make this partition coarser we need greater symmetry relations, and, by the following lemma, greater symmetry groups. To enhance the presentation, all proofs in this paper are deferred to the Appendix.

Lemma 3.2 *Given two groups $G_1, G_2 \subseteq G_I$, we have $G_1 \subseteq G_2$ if and only if $\approx_{G_1} \subseteq \approx_{G_2}$.*

The most important problem, given a symmetry relation $\approx_{G_I}$ is testing membership (i.e., deciding whether $s \approx_{G_I} t$ for any two states $s, t \in \mathcal{S}$). The naive approach of enumerating all permutations of G is hopeless since $|G_I| = |I|!$. The challenge is to find greater symmetry groups for which the membership problem has an effective decision procedure.

One way to compute greater symmetry groups is by composition. Assume for instance that we can identify symmetries within two groups G_{I_1} and G_{I_2} corresponding to disjoint subsets of I i.e., $I_1 \cup I_2 \subseteq I$ and $I_1 \cap I_2 = \emptyset$. Then we can define a composition operator by setting $G_{I_1 \otimes I_2} = \{\pi \in G_I \mid \pi(i) = \pi_k(i)$ if $i \in I_k, \pi_k \in G_{I_k}, k = \{1, 2\}$ and $\pi(i) = i$ otherwise$\}$. Obviously, $G_{I_k} \subseteq G_{I_1 \otimes I_2}$ since each permutation in G_{I_k} is trivially a permutation in $G_{I_1 \otimes I_2}$, for $k = 1, 2$. According to Lemma 3.2, this leads to a larger symmetry relation $\approx_{G_{I_1 \otimes I_2}}$ for which the complexity the membership problem is the sum of the complexities for G_{I_1} and G_{I_2}. This construction shows the advantage of combining independent symmetry reduction techniques for better verification, e.g., by combining heap symmetry and thread symmetry reductions.

3.2 Sorting Criteria Revisited

As mentioned before, given a symmetry relation induced by a permutation group, the membership problem is computationally hard. This fact requires the use of heuristic techniques. The basic idea behind all heuristics used in this paper is to *order* the objects of a state according to some given criterion and use the *sorted* state as a representative for the symmetry equivalence class. We decide that two states are in the same class if and only if their sortings coincide. Notice that this approach is pessimistic in deciding state symmetry

(i.e., two states whose sortings match are symmetric, however some symmetric states may not have coincident sortings).

Let us now consider a total order $<: I \times I$. Formally a *sorting criterion* is a ternary predicate $\xi : \mathcal{S} \times I \times I \to \mathbb{B}$. The intended meaning is that $\xi(s, i, j)$ is true for a state s and two indices i and j, only if the objects indexed by i and j respectively are in the right order. For example, if i and j are entries in a vector of natural numbers v and $\xi(v, i, j) \stackrel{\text{def}}{=} v[i] < v[j]$, then we can say that the vector is sorted when $i < j$ implies $\xi(v, i, j)$, for each $i, j \in I$.

Assuming that t is a sorting of s, there must be a permutation π such that $\pi(s) = t$. Such a permutation is called a *sorting permutation* and is formally defined as follows:

Definition 3.3 Given a state $s \in \mathcal{S}$ and a sorting criterion $\xi : \mathcal{S} \times I \times I \to \mathbb{B}$, a permutation $\pi \in G_I$ is said to be sorting for ξ in s if and only if:

$$\forall i, j \in I.\pi(i) < \pi(j) \Rightarrow \xi(\pi(s), \pi(i), \pi(j))$$

The above definition guarantees neither the existence nor the uniqueness of sorting permutations. Given a state s and a sorting criterion ξ we define $Sort(s, \xi) = \{\pi \in G_I \mid \forall i, j \in I.\pi(i) < \pi(j) \Rightarrow \xi(s, i, j)\}$ be the set of all sorting permutations for ξ in s. These sets enjoy the following monotonicity property, where the implications of sorting criteria are defined point-wise:

Lemma 3.4 *Given two sorting criteria $\xi_1, \xi_2 : \mathcal{S} \times I \times I \to \mathbb{B}$ such that $\xi_1 \Rightarrow \xi_2$ then $Sort(s, \xi_1) \subseteq Sort(s, \xi_2)$ for each $s \in \mathcal{S}$.*

If, for all states $s \in \mathcal{S}$ we have $Sort(s, \xi) \neq \emptyset$, we say that ξ is *well-formed*. The next definition introduces an *invariance* property that needs to be met by a sorting criterion in order to be useful for symmetry reduction purposes.

Definition 3.5 A sorting criterion $\xi : \mathcal{S} \times I \times I \to \mathbb{B}$ is said to be invariant if and only if, for any permutation $\pi \in G_I$ and indices $i, j \in I$ we have $\xi(s, i, j) = \xi(\pi(s), \pi(i), \pi(j))$.

Obviously, a permutation is sorting for an invariant criterion ξ in state s if and only if $\pi(i) < \pi(j) \Rightarrow \xi(s, i, j)$ for any indices $i, j \in I$. The following theorem is the main result of this section. It sets a bound on the complexity of the symmetry problem. To the best of our knowledge, this is also the lowest bound.

Theorem 3.6 *Given two states $s, t \in \mathcal{S}$ and a well-formed invariant criterion $\xi : \mathcal{S} \times I \times I \to \mathbb{B}$ we have $s \approx_{G_I} t$ if and only if there exists $\pi_1 \in Sort(s, \xi)$ and $\pi_2 \in Sort(t, \xi)$ such that $\pi_1(s) = \pi_2(t)$.*

This theorem generalizes our preliminary result in [17]. There an additional condition was imposed on ξ, to reduce the size of $Sort(s, \xi)$ to one:

$$\forall i, j \in I.[\xi(s, i, j) \vee \xi(s, j, i)] \wedge \neg[\xi(s, i, j) \wedge \xi(s, j, i)] \tag{1}$$

In other words, the order induced by the criterion is both strict and total, and thus unique. However finding invariant criteria that meet this requirement is hard, and therefore we shall content ourselves with approximations. In the light of Lemma 3.4, using stronger criteria minimizes the size of the *Sort* sets, improving the time performance of the symmetry decision procedure. On the other hand, when using non-canonical reduction methods, one chooses non-deterministically between two permutations π_1 and π_2 trying to meet the condition of Theorem 3.6. Obviously, having smaller *Sort* sets improves the chance of choosing two matching permutations.

3.3 Heap Symmetry with k-Bounded Thread Symmetry

In this section we present two symmetry reduction methods that use sorting permutations. The present implementation of Bogor combines the two methods, in a way that is formally defined by the $\otimes$ operation, introduced in Section 3.1.

For the purposes of this section we will refine the structure of a state $s = \{o_i\}_{i \in I}$ over an index set I with o_ι as the root object. Assume now that each object o_i is a tuple $\langle t, v_1, \ldots v_n \rangle \in (Type \cup PC) \times Val \times \ldots Val$. An object has an associated type (or a program counter if it is a thread[1]) and carries an any finite number of values. Let $fst(i)$ denote the type (or the program counter), $val(i)$ denote the tuple of values carried by o_i, and $val(i)_j$ denote the selection of the j-th element of $val(i)$. We assume that the set of types, program counters, and values are totally ordered.

In real object-oriented applications objects are organized as a directed graph where objects are nodes and edges represent *pointers* between objects. This graph is sometimes referred to as the *shape graph*. In defining the shape graph of a BIR state, we consider a partition over the set I of indices as $I = \{\iota\} \cup H \cup T$ where $\{\iota\}$, H and T are disjoint sets and:

- H contains the indices of *passive objects*,
- T contains the indices of *active objects*, i.e., *threads*.

Let Σ be a set of pointer variables, or edge labels in our setting. Also, let ϵ be a special symbol, not in Σ. The shape graph of a state is (I, E) where $E \subseteq I \times \Sigma \cup \{\epsilon\} \times I$ is a set of directed labeled edges. As usual, we write $i \xrightarrow{\sigma} j$ for $(i, \sigma, j) \in E$. A path in the graph is a sequence of indices $i = i_1, i_2, \ldots, i_n = j$, where $(i_k, \sigma_k, i_{k+1}) \in E$ for each $1 \leq k < n$. We denote paths by $i \stackrel{w}{\Rightarrow} j$ where $w = \sigma_1 \sigma_2 \ldots \sigma_n$. A node j is said to be reachable from i if and only if there exists a word w over $\Sigma \cup \{\epsilon\}$ such that $i \stackrel{w}{\Rightarrow} j$. We assume that all nodes in the shape graph are reachable from the root node ι, and the shape graph satisfies the following requirements:

[1] We assume without loss of generality that threads only have a program counter instead of a stack of program counters

(i) for no $\sigma \in \Sigma \cup \{\epsilon\}$ and no $k \in I$, $k \xrightarrow{\sigma} \iota$ (i.e., there are no incoming edges to the root node);

(ii) for all $t \in T$, we have $\iota \xrightarrow{\epsilon} t$;

(iii) if $i \xrightarrow{\sigma} j$ and $i \xrightarrow{\tau} k$ for $\sigma, \tau \in \Sigma$ and $j \neq k$ then $\sigma \neq \tau$ (i.e., the graph is deterministic, except for the ϵ edges).

The representation of states as shape graphs is intuitive (Figure 2). Moreover, it is easy to see that the problem of deciding symmetry between states reduces to the graph isomorphism problem. In general, this problem is in NP whereas for deterministic graphs (i.e., no two or more outgoing edges from an object with the same label), it is in P. In our case, the only sources of nondeterminism within the shape graph are the ϵ edges between the root node and the threads (requirement ii above). The intuition behind this is common to garbage collected languages (e.g., Java): passive objects can exist only if they are reachable from the set of program variables (i.e., the root node in our case), whereas active objects can live as long as they still execute, even though they have become unreachable.

Let us now give a (heuristic) solution to the symmetry problem based on sorting criteria (Section 3.2). Assume that ϵ is the neutral element of word concatenation over Σ^* i.e. $w\epsilon = \epsilon w = w$. Given two nodes in the shape graph we define the following set of words:

$$i/j = \{w \in \Sigma^* \mid i \stackrel{w}{\Rightarrow} j\}$$

that label all paths initiating in i and ending in j. Since Σ is finite, we can assume a total order $\prec$: $\Sigma \times \Sigma$ and the lexicographical order induced is denoted by $\prec^*$: $\Sigma^* \times \Sigma^*$. By the fact that $\prec$ is total, we have that $\prec^*$ is also total, and since Σ is finite, each subset S of Σ^* has a minimal element denoted by $\mathit{inf}^* S$.

We say that a node i *dominates* another node j in the shape graph of a state s if all paths reaching j go through i. For an arbitrary node i, we denote by $\mathit{dom}(i)$ the set of nodes it dominates. Then $D_T(s) = \bigcup_{t \in T} \mathit{dom}(t)$ is the set of nodes dominated by a thread.

For an arbitrary node i let $\mathit{ord}(i)$ denote the ordering of its successors i.e., $\langle j_1 \ldots j_n \rangle$ such that $i \xrightarrow{\sigma_k} j_k$ and, for all $k < l$ we have $\sigma_k \prec \sigma_l$. Next, for each node i we define its *signature* to be $\mathit{sign}(i) = \langle \mathit{fst}(i), \mathit{val}(i) \rangle \ \langle \mathit{sign}(j) \rangle_{j \in \mathit{ord}(i)}$, i.e., the word obtained by concatenating its type (or program counter), value and the types and values of all its successors in depth-first order. By $\mathit{sign}_k(i)$, we denote the first k type-value entries in the $\mathit{sign}(i)$ word. The *sign* function can be lifted to work on a set of nodes in a natural way. Since all types and values are totally ordered, we have a lexicographical order on signatures, denoted by $\leq^*$. The $\leq^*$ ordering can be lifted to work on sets of signatures by comparing the least signatures of the sets. Let us denote by $\Theta(i)$ the thread dominators of a node i.

We are now ready to define the sorting criterion that is used in Bogor to deal with symmetry:

$$\xi_k(s,i,j) = \begin{cases} \mathit{inf}^*(\iota/i) \prec^* \mathit{inf}^*(\iota/j) \wedge \\ \quad \mathit{sign}_k(\Theta(i)) \leq^* \mathit{sign}_k(\Theta(j)) & \text{if } i,j \in D_T(s) \\ \mathit{inf}^*(\iota/i) \prec^* \mathit{inf}^*(\iota/j) & \text{otherwise} \end{cases}$$

It can be shown [18] that any two nodes that are not both thread-dominated (the default case), can be totally ordered by the sorting criterion, in the sense of condition (1). The problem caused by the non-determinism in the shape graph is reflected by the first case $i, j \in D_T(s)$. Here, we need to strengthen the sorting criterion by comparing the signatures of the least thread dominators of i and j. Note that by the fact that $\leq^*$ is a lexicographical order, for $k_1 < k_2$, we have $\mathit{sign}_{k_2}(\Theta(i)) \leq^* \mathit{sign}_{k_2}(\Theta(j)) \Rightarrow \mathit{sign}_{k_1}(\Theta(i)) \leq^* \mathit{sign}_{k_1}(\Theta(j))$. Hence $\xi_{k_2} \Rightarrow \xi_{k_1}$. By Theorem 3.6, in combination with Lemma 3.4, this means that we obtain potentially better reduction by increasing the k factor. In general, this increases also the time complexity of computing $\mathit{sign}_k(i)$ by a linear factor. An algorithm that decides, given two nodes i and j, whether $\mathit{inf}^*(\iota/i) \prec^* \mathit{inf}^*(\iota/j)$ is given in [17]. In the next section, we present the kBOTS algorithm implemented in Bogor, that given two threads t_1 and t_2 computes $\mathit{sign}_k(t_1) \leq^* \mathit{sign}_k(t_2)$.

3.4 The kBOTS Algorithm

The sorting criterion defined in the previous section depends on the *observables* of the state from the point of view of each thread in the system. Observables of a thread are defined as: (1) the thread stack height, (2) the program counters, (3) the thread local (primitive) values in the thread stack frames, and (4) the reachable objects from the thread. The computation of the thread symmetry may be expensive in the worst case, however, the computation can be configured so that it only compares up to k-navigations of heap objects; thus, the term k-Bounded Thread Symmetry (kBOTS).

Figure 3 presents the kBOTS algorithm for ordering two threads. The threads are first ordered based on stack heights (begins at line 1), thread counters (begins at line 4) , local (primitive) values in the thread stack frame (line 7-14), because the ordering of these are generally cheap. If the threads cannot be ordered based on those observables, then they are ordered based on their reachable objects. If there are no such reachable objects, i.e., the values of the threads's non-primitive type locals are null, then they are indistinguishable. Thus, they are equivalent.

If there are some reachable objects, then the objects are compared based on the data that are contained in the objects. The comparison is as follows. The first reachable objects are first compared based on their actual types (begins at line 33) and their primitive field values (line 36-43). If they cannot be ordered based on those observables, then navigate to the next unvisited

```
kBOTS(state, t1, t2)
01 result := stackHeightCompare(state, t1, t2)
02 if result ≠ EQUAL then return result
03 result := threadCounterCompare(state, t1, t2)
04 if result ≠ EQUAL then return result
05 for the next local primitive values of t1 and t2, x and
y
06      result := primitiveValueCompare(x, y)
07      if result ≠ EQUAL then return result
08 for the next local non-primitive values of t1 and t2, a
and b
09      result := kBOTS-DFS(a, b, ∅, ∅, 0)
10      if result ≠ EQUAL then return result
11 return EQUAL
end kBOTS

kBOTS-DFS(a, b, seen1, seen2, i)
12 if i == k or a == b return EQUAL
13 if a is in seen1 or a is null return LESS
14 if b is in seen2 or b is null return GREATER
15 seen1 := seen1 + a
16 seen2 := seen2 + b
17 result := typeCompare(a, b)
18 if result ≠ EQUAL then return result
19 for the next field primitive values in a and b, x and y
20      result := primitiveValueCompare(x, y)
21      if result ≠ EQUAL then return result
22 for the next local non-primitive values of a and b, a′
and b′
23      result := kBOTS-DFS(a′, b′, seen1, seen2, i+1)
24      if result ≠ EQUAL then return result
25 return EQUAL
end kBOTS-rec
```

Fig. 3. k-Bounded Thread Symmetry (kBOTS) Algorithm

```
SORT(v)
01 for next object v′ immediately reachable from v do
02      if v′ not marked do
03          mark v′
04          heap := heap[v′ ↦ k]
05          k := k + 1
06          SORT(v′)
end SORT

begin main
07 k := 1
08 heap := [null↦0]
09 for next object v immediately reachable from a global
variable do
10      SORT(v)
11 for next thread p do
12      for next object v immediately reachable from a lo-
cal variable of p do
13          SORT(v)
end main
```

Fig. 4. Heap Objects Sorting Permutations Algorithm

reachable object, and compare them (line 44-51). This recursive comparison is done for all the reachable objects if k is specified to be infinite. The algorithm always terminates because of the nature of the depth-first search algorithm that is used (with or without a bound). To sort all threads, existing sorting algorithms can be used with kBOTS as the comparison function for two threads.

For completeness, Figure 4 presents the heap symmetry algorithm that is used to generate the sorting permutation for heap objects. This algorithm has been presented in [17]. We refer the reader to that paper for a detailed discussion of the algorithm.

In order to illustrate how the heap symmetry with kBOTS works, let us consider the 3-dining philosophers example with the state that is illustrated in Figure 2. Suppose that the ordering on boolean values is that false is less than true. Suppose also that k is infinite. First the threads are ordered as follows. Clearly, P1 is greater than both P2 and P3 based on their location. However, P2 and P3 cannot be ordered based on their stack heights, thread counters or thread local (primitive) values in the stack frame. Thus, we need to compare the reachable objects from P2 and P3. Note that P3 refers to a F1 that is being held by P1 and P2 refers to free forks. Thus, P2 $<$ P3. Therefore, the ordering of threads is P2 $<$ P3 $<$ P1. Once the ordering of the threads is obtained, then we sort the heap objects using Figure 4. We will discuss the usage of the heap-objects-sorted permutation and the threads-sorted permutation in the next section.

The complexity of thread ordering is $O(P \times N + P\ log\ P)$ where N is the number of objects and P the number of threads. To compute the signature

of each thread at most N visits to the objects in the heap are needed. Hence $P \times N$. Then the ordering itself would take no more than $P\ log\ P$ steps with classical sorting algorithms, considering that a comparison of two signatures is atomic.

3.5 Symmetry in Type Extensions

Bogor[21] allows new abstract data structures to be defined as first-class constructs of the BIR modeling language via extensions. One of the obligations of the extension when introducing new abstract data types is to implement an interface to linearize the values of the data types. That is, Bogor requires the new data type value to be converted to a bit-vector.

However, some data types, for example, a key-value table does not have a natural ordering over its elements. A table representation is insensitive to the ordering of the key and value mappings, however, we want to have a unique representation of the table for model checking purpose, if possible.

Our approach for thread symmetry reduction can also be adapted for linearizing these kind of abstract data types. That is, we can, for example, sort the elements of the table by using each object's signature as defined in Subsection 3.3.

4 Collapse Compression

Collapse compression is a reduction technique that gives a significant reduction in the amount of memory needed to store the visited states during finite-state exploration [16,20]. State collapsing is based on the observation that although the number of distinct states in a finite-state exploration can become very large, the number of distinct states of parts of the system such as data in objects or threads and globals, are usually smaller. These parts of the state can be shared across all the visited states that are stored, instead of storing the complete encoding of state every time a new state is visited. For example, consider the 3 dining philosophers example when the system state is the one that is illustrated in Figure 2. When the philosopher holding his left fork takes his right fork, then the only change to the state is the philosopher's right fork and the philosopher's thread. The other part of the state does not change. Therefore, when storing the encoding of the new state, we can reuse the parts of state that does not change from the encoding of the previous state.

The efficiency of a collapse compression algorithm depends on the choice of parts of the state that are collapsed. In this section, we present a collapse compression that takes advantage of the natural structure in object-oriented languages such as Java in order to determine the parts to be collapsed. Furthermore, the collapse compression that we use takes advantage of the symmetry reductions that are described in the previous section.

4.1 Structure Collapsing

Given a state $s = \{o_i\}_{i \in I}$ with a shape graph (I, E) where $I = \{\iota\} \cup H \cup T$, the symmetry reductions presented earlier sort the objects indexed by both H and T. Thus, the objects indexed by H can be numbered from 1 through N for $|H| = N$ that reflects the DFS ordering number of Figure 4. Similarly, threads can be given a number based on the ordering of the thread symmetry. This is a nice property that will be used to efficiently represent states as shown below. We first describe the notations that are used for describing the collapse compression algorithm, and then we describe how the collapse compression is performed.

A bit-vector $\boldsymbol{v} \in BitVector$ is denoted using $\langle$ and $\rangle$. The bit-vector can have the base number annotation as its subscript. For example, both $\langle 11 \rangle_2$ and $\langle 3 \rangle$ denotes a bit-vector containing the integer 3. We use a *comma* (,) to append elements of a bit-vector. For example, $\langle 1, 3 \rangle$ denotes a bit-vector that is obtained by appending $\langle 3 \rangle$ to $\langle 1 \rangle$. Boolean values are represented using only one bit, however, we allow boolean values written inside bit-vectors for presentation, e.g., $\langle \text{true} \rangle$. We also allow other values to be directly written inside the bit-vectors when it is unambiguous to do so. We assume that the number of bits to encode locations and integers are fixed apriori.

A bit-vector pool $\Gamma \in BitVectorPool$ is a function of type $(BitVector \rightharpoonup Int) \times BitVector \rightharpoonup Int \times (BitVector \rightharpoonup Int)$ that is defined as follows:

$$\Gamma(\Delta, \boldsymbol{v}) = \begin{cases} (\Delta(\boldsymbol{v}), \Delta), & \text{if } \boldsymbol{v} \in dom(\Delta) \\ (n, \Delta[\boldsymbol{v} \mapsto n]), & \text{otherwise,} \end{cases}$$

where $n = |\Delta| + 1$. The intuition is that whenever the pool Γ is given a new bit-vector that is not mapped in the table Δ, it then assigns the new bit-vector to the next available integer and returns it along with an updated table that maps the new bit-vector to the integer. Whenever it is given a bit-vector that is mapped in the table, it simply returns the unique integer of the bit-vector and the table.

A values-vector pool $\Lambda \in ValuesVectorPool$ is a function of type $\mathcal{S} \times I \times (BitVector \rightharpoonup Int) \rightharpoonup Int \times (BitVector \rightharpoonup Int)$ that is defined as follows:

$$\Lambda(s, i, \Delta) = \Gamma(\Delta, \langle n_1, \ldots, n_N, m_1, \ldots, m_M \rangle),$$

where $val(i) = (n_1, \ldots, n_N)$, $ord(i) = (j_1, \ldots, j_M)$, $\forall 1 \leq k \leq M . m_k = f(j_k)$,

$$f(l) = \begin{cases} order(O_H, o_l), \text{ if } l \in H \\ order(O_T, o_l), \text{ otherwise,} \end{cases}$$

$order(S, e)$ gives the order number of element e in an ordered-set S, O_H denotes the ordered set of heap objects that are indexed by H, and O_T denotes the ordered set of threads that are indexed by T. This is where we exploit

the ordering number given by the symmetry reductions to refer to the objects and threads.

A thread-vector pool $\Psi \in ThreadVectorPool$ is a function of type $\mathcal{S} \times I \times (BitVector \rightharpoonup Int) \rightharpoonup Int \times (BitVector \rightharpoonup Int)$ that is defined as follows:

$$\Psi(s, i, \Delta) = \Gamma(\Delta', \langle fst(i), n \rangle),$$

where $\Lambda(s, i, \Delta) = (n, \Delta')$.

Given a state $s = \{o_i\}_{i \in I}$ with a shape graph (I, E) where $I = \{\iota\} \cup H \cup T$, $|H| = N$, $|T| = M$, and a bit-vector table Δ, the collapse compression is performed as follows:

(i) We first collapse the global store o_ι by applying Λ, i.e., $\Lambda(s, \iota, \Delta) = (n, \Delta_0)$.

(ii) We then collapse the heap objects indexed by H. Since the objects indexed by H are now ordered from 1 to N, then we can encode the heap as the bit-vector V_H, $V_H = \langle fst(h_1), n_1, \ldots, fst(h_N), n_N \rangle$ where $\forall 1 \leq i \leq N. \Lambda(s, h_i, \Delta_{i-1}) = (n_i, \Delta_i), n_1 = n + 1$, $h_i \in H$, and $order(O_H, o_{h_i}) = i$. We then get the final encoding of the heap by using Δ_N (i.e., $\Gamma(\Delta_N, V_H) = (n_{N+1}, \Delta_{N+1})$). Note that the choice of putting type information in the encoding of heap instead of the encoding of each object is to allow sharing between object bit-patterns although their types are different.

(iii) Next, we collapse the threads indexed by T. Since the threads indexed by T are now ordered from 1 to M, we can encode the threads as a bit-vector V_T, $V_T = \langle m_1, \ldots, m_M \rangle$ where $\forall 1 \leq i \leq M. \Psi(s, t_i, \Delta_{N+i}) = (m_i, \Delta_{N+i+1}), m_1 = n_{N+1} + 1, t_i \in T$, and $order(O_T, o_{t_i}) = i$. We then get the final encoding of the threads by using Δ_{N+M+1} (i.e., $\Gamma(\Delta_{N+M+1}, V_T) = (m_{M+1}, \Delta_{N+M+2})$).

(iv) Last, the encoding of the state is the bit-vector $\langle n, n_{N+1}, m_{M+1} \rangle$ that consists of the integer representations of the global store, the heap, and the threads. Therefore, the final encoding of the state s is $\Gamma(\Delta_{N+M+2}, \langle n, n_{N+1}, m_{M+1} \rangle) = (n, \Delta_{N+M+3})$.

Note that we use only one bit-vector pool for simplicity of the presentation. In general, several bit-vector pools may be used. Moreover, the Γ may further recursively divides the given bit-vector into small chunks of bit-patterns, e.g., by limiting the maximum number of bits that a bit-vector can have.

Figure 5 illustrates the process of collapsing the state at Figure 2 with $\Delta = \emptyset$. The column **n** denotes the unique number of the bit-vector in the **Bit-vector column**, the bit-vector table is propagated down the rows as Γ is applied. We describe some of the collapsing cases briefly.

Since there is no global variable in the dining philosopher example, then, the global variables bit-vector is empty (after ϵ edges are removed). Since the ordering of objects in Figure 2 is $\mathsf{F3} < \mathsf{F2} < \mathsf{F1}$, then the heap is encoded as the bit-vector containing the encoding number of F3, F2, and F1, respectively. Similarly, since the ordering of the threads in Figure 2 is $\mathsf{P2} < \mathsf{P3} < \mathsf{P1}$, then

Structure	Bit-vector	n
Globals	$\langle\rangle$	1
F3	$\langle$false$\rangle$	2
F2	$\langle$false$\rangle$	2
F1	$\langle$true$\rangle$	3
Heap	$\langle \tau_{Fork}, 2, \tau_{Fork}, 2, \tau_{Fork}, 3\rangle$	4
P2 Store	$\langle 1, 2\rangle$	5
P2	$\langle$loc0,5$\rangle$	6
P3 Store	$\langle 2, 3\rangle$	7
P3	$\langle$loc0,7$\rangle$	8
P1 Store	$\langle 3, 1\rangle$	9
P1	$\langle$loc1,9$\rangle$	10
Threads	$\langle 6, 8, 10\rangle$	11
State	$\langle 1, 4, 11\rangle$	12

Fig. 5. Collapsed state of Figure 2

Structure	Bit-vector	n
Globals	$\langle\rangle$	1
F2	$\langle$false$\rangle$	2
F3	$\langle$true$\rangle$	**3**
F1	$\langle$true$\rangle$	3
Heap	$\langle \tau_{tork}, 2, \tau_{Fork}, 3, \tau_{Fork}, 3\rangle$	**13**
P3 Store	$\langle 1, 2\rangle$	5
P3	$\langle$loc0,5$\rangle$	6
P2 Store	$\langle 3, 2\rangle$	**14**
P2	$\langle$loc0,14$\rangle$	**15**
P1 Store	$\langle 2, 3\rangle$	7
P1	$\langle$loc2,7$\rangle$	**16**
Threads	$\langle 6, 15, 16\rangle$	**17**
State	$\langle 1, 13, 17\rangle$	**18**

Fig. 6. Collapsed state of Figure 2 and after P1 takes its right fork

the threads are encoded as the bit-vector containing the encoding number of P2, P3, and P1, respectively.

Figure 6 illustrates the collapsing process of the state at Figure 2 after P1 takes its right fork with the bit-vector table resulting from the process in Figure 5. In this state, the orderings are $\mathsf{P3} < \mathsf{P2} < \mathsf{P1}$ and $\mathsf{F2} < \mathsf{F1} < \mathsf{F3}$. We use a **bold** font to emphasize the differences from Figure 5. Notice that many bit-vectors from the previous state are reused.

5 Preliminary Results

The heap symmetry, k-Bounded Thread Symmetry, and the collapse algorithms presented in this paper have been implemented in Bogor[21]. In order to assess the effectiveness of our approach, we have run several model checks using our approaches and known algorithms for thread symmetry reduction and state compression. There are three modes of thread symmetry reductions: (1) no thread symmetry reduction (NTS) (2) thread symmetry reductions using program counters (PCS) used in [1], and (3) thread symmetry reductions using infinitely-bounded thread symmetry ($\infty-$BOTS). Mode (2) is equivalent to the approach in [17], which are the most effective symmetry reductions for dynamic systems developed to date. There are also three modes of state compressions: (1) no compression (NC), (2) state compression using the GZIP algorithm (GZIP), and (3) collapse compression (COLL). We ran the test cases using the combinations of the thread symmetry reductions and state compressions. We use heap symmetry reduction in all of the experiments.

All experiments were performed on an Intel Pentium 4 2.4 GHz machine with 1 Gb memory, using the Sun J2SE 1.4.1 platform. In the experiment tables that we present below, the amount of memory at the end of each search (m) is given in kilobytes or megabytes (Mb); the amount of time for each search is given in seconds ($x.y$), or hours (h:m:s); the number of states (s) and the number of seen states during the search (seen) is given directly. There

N		3			5			7			10		
		NTS	PCS	∞-BOTS	NTS	PCS	∞-BOTS	NTS	PCS	∞-BOTS	NTS	PCS	∞-BOTS
	s	36	19	14	393	250	160	4287	3337	2207	154451	137863	99976
	seen	41	21	15	1024	659	410	17394	13683	8937	961681	860763	623240
NC	m	324	320	320	358	371	350	1.5Mb	1.2Mb	866	52.6Mb	48.1Mb	33Mb
	t	.16	.12	.11	1.09	.85	.69	13.28	10.81	7.86	0:37:15	0:17:02	0:11:19
GZIP	m	323	320	320	340	319	302	1.1Mb	910	678	32.3Mb	30.2Mb	20Mb
	t	.2	.15	.15	2.06	1.49	.98	25.79	21.78	15.59	0:31:25	0:36:50	0:19:00
COLL	m	322	321	321	321	350	338	856	706	548	22Mb	20.8Mb	13.2Mb
	t	.15	.12	.12	1.04	.83	.68	12.08	9.85	7.37	0:24:08	0:19:55	0:10:25

Table 1
Results for Dining Philosophers Example

are systems used for the experiments: dining philosophers, bounded buffer, and ordered list (from [17]). All of the systems are available from the Bogor website [22]. We describe the experiments in the following sub-sections.

5.1 Dining Philosophers

The dining philosophers test case is the one presented earlier. We check for deadlocks in the model with 3, 5, 7, and 10 philosophers (N). Table 1 presents the collected data. Bogor fully explored the states of each test case and it found only one unique deadlock state in each of the test case.

In general, ∞-BOTS with collapse compression gave the best reduction in terms of the number of distinct visited states, the space required to store those states, and the time to perform the search. For example, in the case with 10 philosophers, the search was almost four times faster and used one fourth of memory as compared to the search without using any thread symmetry reduction and state compression.

The search with the GZIP state compression required more time than collapse compression, yet, it consumes more memory. This can be observed clearly when there are 7 or 10 dining philosophers. The ∞-BOTS algorithm always gave a better reduction than using process counters. For example, in the case with 10 philosophers, ∞-BOTS search required half the time of the PC search and used 36% less space.

5.2 Bounded Buffer

This system is similar to the dining philosophers. Instead of locking resources, however, the threads pass messages (as objects) via buffers. The threads and the buffers form a ring to pass messages circularly. The size of the buffers is bounded when the buffers are created. Threads can only take messages from non-empty buffers. If the buffers are empty, then the threads are blocked. Conversely, threads can only add messages to non-full buffers. If the buffers are full, then the threads are blocked. The buffers use lock mechanisms similar to Java monitors for enforcing the described synchronization conditions.

Table 2 presents the result of the experiments. N refers to the number of bounded buffers and, because of the ring structure, the number of threads.

N, M		3, 1			4, 2		
		NTS	PCS	∞-BOTS	NTS	PCS	∞-BOTS
	s	76954	35543	33432	1134990	568882	517212
	seen	128981	59895	56231	2903779	1472489	1339315
NC	m	26.5Mb	13.0Mb	12.0Mb	469.6Mb	233.6Mb	213.7Mb
	t	0:03:0	0:01:25	0:01:21	4:10:46	0:44:48	0:43:11
GZIP	m	18.0Mb	8.6Mb	8.2Mb	299.2Mb	147.9Mb	135.8Mb
	t	0:05:41	0:02:43	0:02:32	3:37:51	1:15:59	1:06:50
COLL	m	7.5Mb	4.2Mb	4.0Mb	107.0Mb	56.9Mb	53.2Mb
	t	0:02:41	0:01:15	0:01:11	4:05:37	0:46:23	0:37:23

Table 2
Results for Bounded-Buffer Example

N, MAX		3, 3		4, 3	
		PCS	∞-BOTS	PCS	∞-BOTS
	s	105773	105218	4126383	4014937
	seen	193082	191980	11285682	10972913
GZIP	m	20.82 Mb	20.73 Mb	876.11 Mb	855.49 Mb
	t	0:12:43	0:12:11	10:43:13	10:14:0
COLL	m	8.32Mb	8.3Mb	311.29Mb	308.43Mb
	t	0:2:38	0:2:41	5:6:33	4:58:58

Table 3
Results for Ordered-List Example

M refers to the number of messages (objects) that are passed. Searches with collapse compression only consumed 25% to 33% of the space required without state compression. While GZIP compression offered some reductions, it was more than twice as consumptive of memory than our collapse compression. Furthermore, GZIP compression introduced more runtime overhead than collapse compression. ∞-BOTS yielded only marginal improvement (9%) in terms of state reduction, memory and time over PCS for the small number of threads (3 and 4) considered.

5.3 Ordered List

The ordered list example is a linked list ordered by key nodes that is accessed concurrently by an arbitrary number of updater threads (N). Every such thread inserts values in the list. Each node has a lock used by the threads to synchronize. Each thread needs to hold the lock of the current and next node in the list in order to preserve the structure. The number of insert actions done by each updater is bounded (MAX).

Table 3 presents the result of the ordered-list experiments. We omit the comparison with (NTS) because it has been done in [17]. As with the Bounded-Buffer, collapse compression outperformed GZIP significantly in terms of both space and time. Although ∞-BOTS did not give significant reduction over the PCS, there was a significant upward trend in the percentage reduction as the number of updater threads increased (3 threads had a 0.02% reduction and 4 threads had a 2.5% reduction). We plan to run additional experiments to understand the scaling of reductions with system size.

6 Related Work

Symmetries have been proposed by a number of researchers [1,4,5,10,17,19] as a means of reducing the cost of model checking. Here we only compare approaches that address issues in dynamic and concurrent programs (e.g., Bosnacki et. al. [1] and our previous work [17]); these subsume the more static approaches in the literature for the most part.

Bosnacki et. al. introduced a form of thread symmetry reduction that is achieved by ordering program counters when ordering threads. Although the idea is simple, the reduction is very effective. Our previous work on heap symmetry reduction takes advantage of the fact that the location of objects in modern programming languages such as Java are irrelevant to the programs written in that language. Thus, when checking a Java program, for example, we can find state symmetries based on heap objects by permuting the locations of the objects. In that work, we gave an efficient algorithm to find a canonical representation of deterministic heap object diagrams.

Our current work extends the above mentioned work by providing better heuristics for deciding ordering of elements of a state. We extend the work by Bosnacki et. al. by adding better heuristics than ordering based on only thread program counters. We also extend our previous work by providing heuristics to find symmetries even in the presence of non-deterministic heap object diagrams. This had not been handled previously, and it is very useful in an extensible model checker such as Bogor where users can define new abstract data types. For example, our method can be used to order elements of a set of objects.

Another approach to heap symmetry reduction is embodied in JPF [20], but it is known that the approach does not give rise to a canonical state representation. Our approach on heap symmetry reduction that is derived from our previous work [17] can find a canonical state representation as mentioned previously.

Collapse compression has been used in Spin [16] and JPF [20] and has been shown to give significant space reductions when model checking as confirmed by our approach. The main novelty of our work is that we take advantage of the information provided by the heap symmetry and the thread symmetry reductions to drive compression of the complex data state that arises in object-oriented programs.

7 Conclusion and Future Work

This paper has proposed the extension of two existing frameworks for reducing the memory required when model checking software. We present a framework that integrates approaches to thread and heap symmetry. The resulting framework is tunable to allow for approximation of symmetry calculations to trade model checking overhead cost with the degree of reduction. The information

calculated by this framework can be used to drive the compression of state encodings that exploit the shared structure of different program states. We present preliminary data that suggests the benefit of these reductions.

While this work is mature from a foundational perspective, we believe that additional empirical study is warranted. We would like to increase the breadth of our empirical evaluation to understand the sensitivity of our methods to program structure. While it is clear that state-of-the-art symmetry reductions can yield dramatic space and time savings in model checking, the degree of further reduction that is possible remains unclear. We would like to understand the extent to which the existing reduction frameworks are reaching the point of diminishing returns. We plan to further adapt our symmetry framework to explore this question.

References

[1] D. Bosnacki, D. Dams, and L. Holenderski. Symmetric spin. In *International Journal on Software Tools for Technology Transfer*. Springer-Verlag, 2002.

[2] G. Brat, K. Havelund, S. Park, and W. Visser. Java PathFinder – a second generation of a Java model-checker. In *Proceedings of the Workshop on Advances in Verification*, July 2000.

[3] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[4] E. M. Clarke, E. A. Emerson, S. Jha, and A. P. Sistla. Exploiting symmetries in temporal logic model checking. In *Proc. 5th International Conference on Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 147–158. Springer-Verlag, 1998.

[5] E. M. Clarke, T. Filkorn, and S. Jha. Exploiting symmetries in temporal logic model checking. In *Proc. 5th International Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[6] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera : Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, June 2000.

[7] C. Demartini, R. Iosif, and R. Sisto. dspin : A dynamic extension of SPIN. In *Theoretical and Applied Aspects of SPIN Model Checking (LNCS 1680)*, Sept. 1999.

[8] W. Deng, M. Dwyer, J. Hatcliff, G. Jung, and Robby. Model-checking middleware-based event-driven real-time embedded software (extended version). In *Proceedings of the 1st International Symposium on Formal Methods for Components and Objects*, 2002. (to appear).

[9] M. B. Dwyer, J. Hatcliff, V. R. Prasad, and Robby. Using static and dynamic escape analysis to enable model reductions in model-checking concurrent object-oriented programs. Technical Report SANTOS-TR2003-1, Kansas State University, Feb. 2003. (submitted for publication).

[10] E. Emerson and C. Jutla. Symmetry and model checking. In *Proc. 5th International Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[11] C. Flanagan and S. Qadeer. Thread-modular model checking. In *Proceedings of Tenth International SPIN Workshop*, volume 2648 of *Lecture Notes in Computer Science*. Springer-Verlag, may 2003.

[12] J. Hatcliff, W. Deng, M. Dwyer, G. Jung, and V. Prasad. Cadena: An integrated development, analysis, and verification environment for component-based systems. In *Proceedings of the 25th International Conference on Software Engineering*, 2003.

[13] M. Henessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1):137–161, 1985.

[14] T. A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In *Proc. 15th International Conference on Computer Aided Verification*, 2003.

[15] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages*, pages 58–70, Jan. 2002.

[16] G. J. Holzmann. State compression in spin: Recursive indexing and compression training runs. In *Proceedings of Third International SPIN Workshop*, Apr. 1997.

[17] R. Iosif. Symmetry reduction criteria for software model checking. In *Proceedings of Ninth International SPIN Workshop*, volume 2318 of *Lecture Notes in Computer Science*, pages 22–41. Springer-Verlag, Apr. 2002.

[18] R. Iosif. Symmetry reductions for model checking of concurrent dynamic software. *International Journal on Software Tools for Technology Transfer*, 2002.

[19] C. N. Ip and D. L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1/2):47–75, Aug. 1996.

[20] F. Lerda and W. Visser. Addressing dynamic issues of program model checking. In M. B. Dwyer, editor, *Proceedings of Eigth International SPIN Workshop*, volume 2057 of *Lecture Notes in Computer Science*, pages 80–102. Springer-Verlag, may 2001.

[21] Robby, M. B. Dwyer, and J. Hatcliff. Bogor: An extensible and highly-modular model checking framework. In *Proceedings of the 4th Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2003. (to appear).

[22] Bogor Website. `http://bogor.projects.cis.ksu.edu`, 2003.

A Appendix

Proof of Lemma 3.2: "⇒" This direction is trivial. "⇐" Assume $s \approx_{G_1} t$ for some states $s, t \in \mathcal{S}$. By definition, there exists $\pi_1 \in G_1$ such that $\pi_1(s) = t$. Since $\approx_{G_1} \subseteq \approx_{G_2}$ we have $s \approx_{G_2} t$, hence there exists $\pi_2 \in G_2$ such that $\pi_2(s) = t = \pi_1(s)$. Since the choice of s is arbitrary we have $\pi_1(s) = \pi_2(s)\ \forall s \in \mathcal{S}$, hence $\pi_1 = \pi_2 \in G_2$. So $G_1 \subseteq G_2$. □

Proof of Lemma 3.4: Let $\pi \in Sort(s, \xi_1)$. Then for some $i, j \in I$ we have $\pi(i) < \pi(j) \Rightarrow \xi_1(\pi(s), \pi(i), \pi(j)) \Rightarrow \xi_2(\pi(s), \pi(i), \pi(j))$. Hence $\pi \in Sort(s, \xi_2)$. □

Proof of Theorem 3.6: "⇒" If $s \approx_{G_I} t$ there exists a permutation $\pi \in G_I$ such that $\pi(s) = t$. Since ξ is well-formed there exists a permutation $\pi_1 \in Sort(s, \xi)$. Let π_2 be the permutation $\pi_1 \circ \pi^{-1}$. We show that $\pi_2 \in Sort(t, \xi)$. For any indices i and j we have:

$$\begin{aligned}
\pi_2(i) < \pi_2(j) &\iff \pi_1(\pi^{-1}(i)) < \pi_2(\pi^{-1}(j)) \\
&\Rightarrow \ \xi(\pi_1(s), \pi_1(\pi^{-1}(i)), \pi_2(\pi^{-1}(j))) \\
&= \ \xi(\pi_1(\pi^{-1}(t)), \pi_1(\pi^{-1}(i)), \pi_2(\pi^{-1}(j))) \\
&= \ \xi(\pi_2(t), \pi_2(i), \pi_2(j))
\end{aligned}$$

Hence $\pi_2 \in Sort(t, \xi)$ and $\pi_1(s) = \pi_2(t)$. "⇐" $s \approx_{G_I} t$ since $\pi_2^{-1} \circ \pi_1(s) = t$.□